CodeNativeCompiler

January 15, 2018

1 Introduction

In normal world of computational coding we use high level languages. But even this level of abstraction is not enough. So, for even simple work we need to write around 100 LOC and also need to repeat the same for different projects. This problem even becomes huge on industrial scale where employees waste their significant amount of time and effort.

On top of that computing languages act as a barrier for native language speakers. One is expected to have a good grip over english in order to write even the simplest of code. This is a major roadblock faced by developers and innovators who do not undersatnd english.
So what is the solution? Developing separate compilers for every native language? No. Actually it is not necessary, In this paper we are trying to solve above mentioned problems which will be even applicable on industrial level.
The key idea here is to make a simple job specific multi native-languages to multi-language converter over a pre-established computer language.

2 Literature Review[1][2]

To develop this utility we need to understand how compilers works and introduce a set of modifications in that approach to make it more robust.

2.1 Lexical Analysis

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters in to meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form (token-name,attribute-value) that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry Is needed for semantic analysis and code generation .

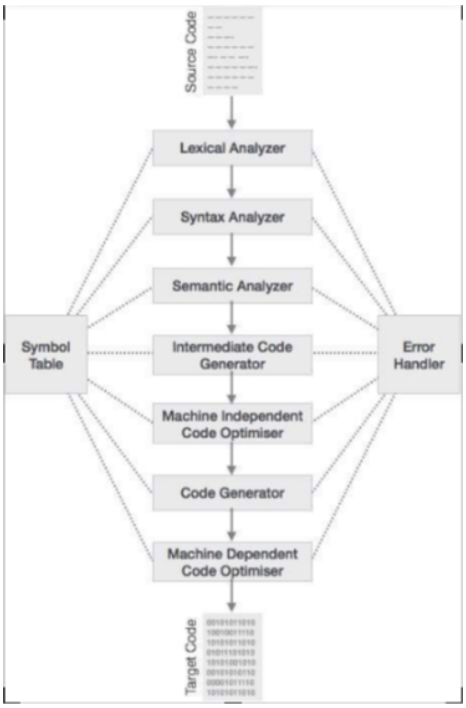

Figure 1: Traditional Way of Compiler Design [1]

2.2 Syntax Analysis

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

2.3 Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

2.4 Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

2.5 Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

2.6 Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

3 Our Approach

So, Our solution for the problem mentioned is just to add a simple job specific multi native-languages to multi-language converter over a pre-established computer language. Here, multiple target-languages are used just for industrial purpose as we may be working on different languages for the same job.

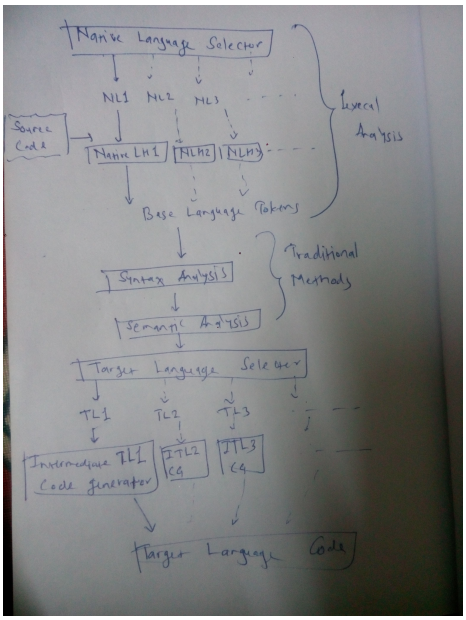

Figure 2: Complete Block diagram of Our Approach

We have chosen GUI as our job, JAVA as our pre-established computer language and english as base language for demonstrating and implementing our approch.

3.1 Defining the Grammar

The very step of our approach is to define the simple Grammar rules for our language converter. The Grammar must be defined in such a way that,

- It solves the problem of large LOC for a simple task.
- The resultant language should be straightforward and easily understandable by the developer.

For example:
To make a simple box in Java a code requires is as follows:

```
import java.awt.Color;
import java.awt.Component;
import java.awt.Font;
import javax.swing.*;
public class Simple2 {
       JFrame f;
       Simple2(){
       f=new JFrame();//creating
           instance of JFrame
JLabel jl0 = new JLabel(
    "firstname");
jl0.setBounds(10,10,300,350);
jl0.setForeground(Color.black);
jl0.setFont(new Font("Serif",
    Font.BOLD,0));
f.add(jl0);
JTextField jt0 = new
    JTextField(
 jt0.setBounds(350,10,700,350);
jt0.setForeground(Color.black);
jt0.setFont(new Font("Serif",
    Font.BOLD,0));
f.add(jt0);
JLabel jl1 = new JLabel(
    "lastname");
jl1.setBounds(10,400,300,750);
jl1.setForeground(Color.black);
jl1.setFont(new Font("Serif",
    Font.BOLD,0));
f.add(jl1);
JTextField jt1 = new
    JTextField(                          );
 jt1.setBounds(350,400,700,750);
jt1.setForeground(Color.black);
jt1.setFont(new Font("Serif",
    Font.BOLD,0));
f.add(jt1);
JButton jb0 = new JButton(
    "submit");
jb0.setBounds(10,800,300,1050);
jb0.setForeground(Color.black);
jb0.setFont(new Font("Serif",
    Font.BOLD,0));
f.add(jb0);
f.setSize(400,500);//400 width and
    500 height
f.setLayout(null);//using no
    layout managers
f.setVisible(true);//making the
    frame visible
}
       public static void
           main(String[] args) {
              new Simple2();
       }
 }
```

but by using converter it can be:

```
                    First   Name
    10 10 300 350 0 0 0
                  First   Name
    350 10 700 350 0 0 0
                    Last    Name
    10 400 300 750 0 0 0
                  Last   Name
                );
    350 400 700 750 0 0 0
           Submit   10 800 300
    1050 0 0 0
```

So, to reduce code complexity se-

lecting the grammar plays the key role here. As you can see clear difference between the above two codes. The latter one is more readable, short and simple. One can even define different grammar to even make the code more understandable. So, It may help in debugging also.

The idea is to define your own grammar which will be more comfortable for you and will save time and cost.

3.2 Selecting Native Languages

Now the task is to find out which languangs you want to add as native languages. These are the languages in which actual user will code. So, this may vary from place to place and the actual users. The good news is, we can add new native languages at any time with adding a simple language handler.

3.3 Language Handler

Language handler can be seen as tokenizer which converts lexemes of that language to the base language token. This part makes the difference, by selecting one base language it is just a task of making one compiler and other language handlers works virtually as a compiler for that new Language.

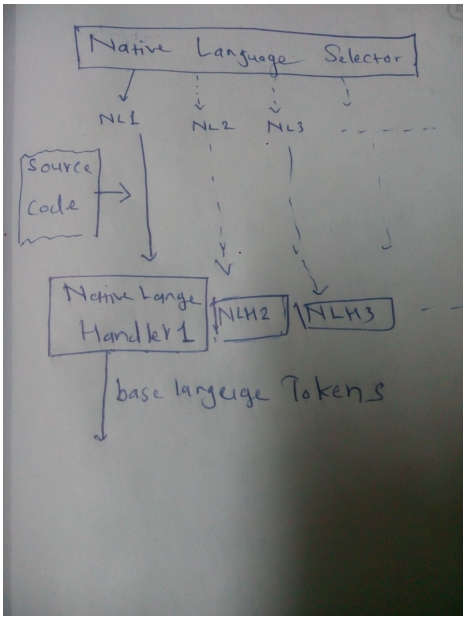

Figure 3: Block diagram of Lexical Analysis

3.4 Syntax Analyser

Now, as we get the base language tokens we simply use traditional methods of compiler design for the further phases.

As in first step we defined our grammar, using that we will make Action and GOTO tables to check the syntax.

From this step it is totally dependent on designer and the Grammar to choose the correct technique for this and next phases. We used LR(1) parser in our implementation.

3.5 Multiple-Target Language Code Generator

This phase is required at industrial level where multiple products are made on multiple languages. So, in place of repeating all the above processes for the same job for different target language, we can simply add the last phase to generate required target code. This task is time and effort consuming, but will be beneficial in long terms. And we can always add new target language by adding only one module.

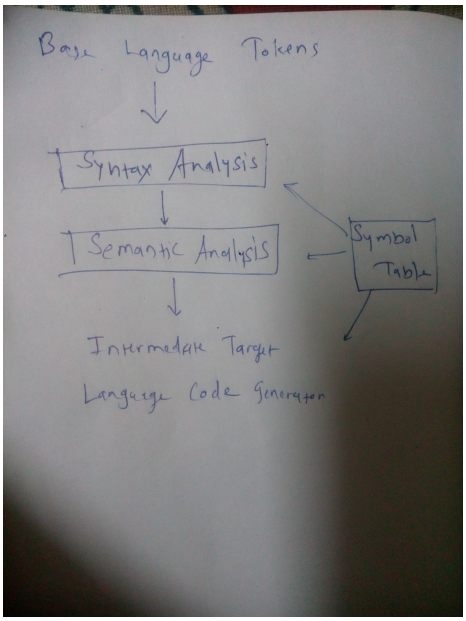

Figure 4: Block diagram of middle phase

4 Summary

We have discussed a simple way to remove language dependancy and complexity in learning high level language of a native developer so anyone without knowing coding languages or even english could code and develope a spe-

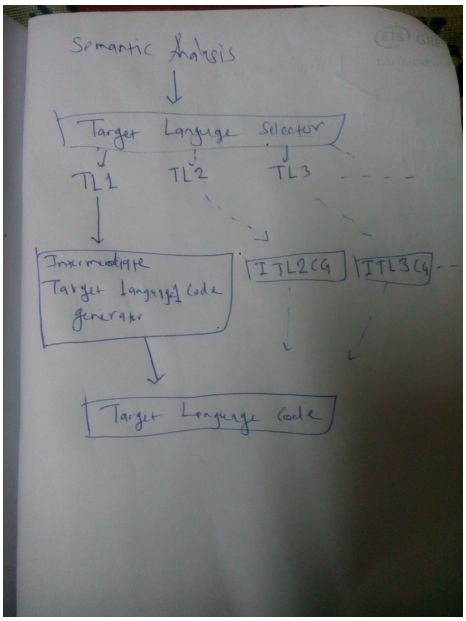

Figure 5: Block diagram of Multiple Target Code Generation

cific task.

5 Future Work

- This same principle can be used to form a modern approach to design a compiler as it will allow mutiple input languages for a single compiler and can add or remove with very less efforts.

- If we could detect the language user is typing, we can develop dynamic Native language selector and language handler which can be used to replace above mentioned lexical phase to dynamic Lexical Analysis.

- Our Approach talks about job specific enhancement of Compiler but, it can be made to multi-job specific or even job independent enhancements of Compiler.

References

[1] Alfred Aho, Jeffrey Ullman, Monica S. Lam, and Ravi Sethi *Compilers: Principles, Techniques, and Tools* 1986.

[2] Tutorials Point: Compiler Design Tutorial *tutorialspoint.com*